

On Compressing Permutations and Adaptive Sorting ^{*}

Jérémy Barbay Gonzalo Navarro

Dept. of Computer Science, University of Chile

Abstract

Previous compact representations of permutations have focused on adding a small index on top of the plain data $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$, in order to efficiently support the application of the inverse or the iterated permutation. In this paper we initiate the study of techniques that exploit the compressibility of the data itself, while retaining efficient computation of $\pi(i)$ and its inverse. In particular, we focus on exploiting *runs*, which are subsets (contiguous or not) of the domain where the permutation is monotonic. Several variants of those types of runs arise in real applications such as inverted indexes and suffix arrays. Furthermore, our improved results on compressed data structures for permutations also yield better adaptive sorting algorithms.

1 Introduction

Permutations of the integers $[1..n] = \{1, \dots, n\}$ are not only a fundamental mathematical structure, but also a basic building block for the succinct encoding of integer functions [MR04], strings [Kär99, GMR06, GV06, ANS06, MN07, CHSV08], binary relations [BHMR07], and geometric grids [BLNS09], among others. A permutation π can be trivially encoded in $n \lceil \lg n \rceil$ bits, which is within $\mathcal{O}(n)$ bits of the information theory lower bound of $\lg(n!)$ bits, where $\lg x = \log_2 x$ denotes the logarithm in base two.

In most of those applications, efficient computation is required for both the value $\pi(i)$ at any point $i \in [1..n]$ of the permutation, and for the position $\pi^{-1}(j)$ of any value $j \in [1..n]$ (i.e., the value of the inverse permutation). The only alternative we are aware of to storing explicitly both π and π^{-1} is by Munro et al. [MRRR03], who add a small structure over the plain representation of π so that, by spending $\epsilon \lg n$ extra bits, any $\pi^{-1}(j)$ can be computed in time $\mathcal{O}(1/\epsilon)$. This is extended to any positive or negative power of π , $\pi^k(i)$. They give another solution using $\mathcal{O}(n)$ extra bits and computing any $\pi^k(j)$ in time $\mathcal{O}(\lg n / \lg \lg n)$.

The lower bound of $\lg(n!)$ bits yields a lower bound of $\Omega(n \lg n)$ comparisons to sort such a permutation in the comparison model, in the worst case over all permutations of n elements. Yet, a large body of research has been dedicated to finding better sorting algorithms which can take advantage of specificities of each permutation to sort. Some examples are permutations composed of a few sorted blocks [Man85] (e.g., $(1, 3, 5, 7, 9, \mathbf{2, 4, 6, 8, 10})$ or $(6, 7, 8, 9, 10, \mathbf{1, 2, 3, 4, 5})$), or permutations containing few sorted subsequences [LP94] (e.g., $(1, \mathbf{6, 2, 7, 3, 8, 4, 9, 5, 10})$). Algorithms performing possibly $o(n \lg n)$ comparisons on such permutations, yet still $\mathcal{O}(n \lg n)$ comparisons in

^{*}Partially funded by Fondecyt Grant 1-110066, Chile. An early partial version of this paper appeared in *STACS* [BN09].

the worst case, are achievable and preferable if those permutations arise with sufficient frequency. Other examples are classes of permutations whose structure makes them interesting for applications: see the seminal paper of Mannila [Man85], and the survey of Moffat and Petersson [MP92] for more details.

Each sorting algorithm in the comparison model yields an encoding scheme for permutations: the result of all comparisons performed uniquely identifies the permutation sorted, and hence encodes it. Since an adaptive sorting algorithm performs $o(n \lg n)$ comparisons on a class of “easy” permutations, each adaptive algorithm yields a *compression scheme* for permutations, at the cost of losing a constant factor on the complementary class of “hard” permutations. Yet such compression schemes do not necessarily support efficiently the computation of value $\pi^{-1}(j)$ of the inverse permutation for an arbitrary value $j \in [1..n]$, or even the simple application of the permutation, $\pi(i)$.

This is the topic of our study: the interplay between adaptive sorting algorithms and compressed representation of permutations that support efficient application of $\pi(i)$ and $\pi^{-1}(j)$. In particular we focus on classes of permutations that can be decomposed into a small number of *runs*, that is, monotone subsequences of π , either contiguous or not.

Our results include compressed representations of permutations whose space and time to compute any $\pi(i)$ and $\pi^{-1}(j)$ are proportional to the *entropy* of the distribution of the sizes of the runs. As far as we know, this is the first compressed representation of permutations with similar capabilities.

We also develop the corresponding sorting algorithms, which in general refine the known complexities to sort those classes of permutations: While there exist sorting algorithms taking advantage of the number of runs of various kinds, ours take advantage of their size distribution and are strictly better (or equal, at worst).

Finally, we obtain a representation for strings that improves upon the state of the art [FMMN07, GRR08] in the average case, while retaining their space and worst-case performance for operations access, rank, and select.

At the end of the article we describe some applications where the class of permutations compressible with the techniques we develop here naturally arise, and conclude with a more general perspective on the meaning of those results and the research directions they suggest.

2 Basic Concepts and Previous Work

2.1 Entropy

We define the *entropy* of a distribution [CT91], a measure that will be useful to evaluate compressibility results.

Definition 1 *The entropy of a sequence of positive integers $X = \langle n_1, n_2, \dots, n_r \rangle$ adding up to n is $\mathcal{H}(X) = \sum_{i=1}^r \frac{n_i}{n} \lg \frac{n}{n_i}$. By concavity of the logarithm, it holds that $(r-1) \lg n \leq n\mathcal{H}(X) \leq n \lg r$ and that $\mathcal{H}(\langle n_1, n_2, \dots, n_r \rangle) > \mathcal{H}(\langle n_1+n_2, \dots, n_r \rangle)$.*

Here $\langle n_1, n_2, \dots, n_r \rangle$ is a distribution of values adding up to n and $\mathcal{H}(X)$ measures how even is the distribution. $\mathcal{H}(X)$ is maximal ($\lg r$) when all $n_i = n/r$ and minimal ($\frac{r-1}{n} \lg n + \frac{n-r+1}{n} \lg \frac{n}{n-r+1}$) when they are most skewed ($X = \langle 1, 1, \dots, 1, n-r+1 \rangle$).

This measure is related to entropy of random variables and of sequences as follows. If a random variable P takes the value i with probability n_i/n , for $1 \leq i \leq r$, then its entropy is

$\mathcal{H}(\langle n_1, n_2, \dots, n_r \rangle)$. Similarly, if a string $S[1..n]$ contains n_i occurrences of character c_i , then its empirical zero-order entropy is $\mathcal{H}_0(S) = \mathcal{H}(\langle n_1, n_2, \dots, n_r \rangle)$.

$\mathcal{H}(X)$ is then a lower bound to the average number of bits needed to encode an instance of P , or to encode a character of S (if we model S statistically with a zero-order model, that is, ignoring the context of characters).

2.2 Huffman Coding

The Huffman algorithm [Huf52] receives frequencies $X = \langle n_1, n_2, \dots, n_r \rangle$ adding up to n , and outputs in $\mathcal{O}(r \lg r)$ time a prefix-free code for the symbols $[1..r]$. If ℓ_i is the bit length of the code assigned to the i th symbol, then $L = \sum \ell_i n_i$ is minimal. Moreover, $L < n(1 + \mathcal{H}(X))$. For example, given $S[1..n]$ over alphabet $[1..r]$, with symbol frequencies X , one can compress S by concatenating the codewords of the successive symbols $S[i]$, achieving total length $L < n(1 + \mathcal{H}_0(S))$. (One also has to encode the usually negligible codebook of $\mathcal{O}(r \lg r)$ bits.)

Huffman's algorithm starts with a forest of r leaves corresponding to the frequencies $\{n_1, n_2, \dots, n_r\}$, and outputs a binary trie with those leaves, in some order. This so-called *Huffman tree* describes the optimal encoding as follows: The sequence of left/right choices (interpreted as 0/1) in the path from the root to each leaf is the prefix-free encoding of that leaf, of length ℓ_i equal to the leaf depth.

A generalization of this encoding is multiary Huffman coding [Huf52], in which the tree is given arity t , and then the Huffman codewords are sequences over an alphabet $[1..t]$. In this case the algorithm also produces the optimal code, of length $L < n(1 + \mathcal{H}(X)/\lg t)$.

2.3 Succinct Data Structures for Sequences

Let $S[1..n]$ be a sequence of symbols from the alphabet $[1..r]$. This includes bitmaps when $r = 2$ (where, for convenience, the alphabet will be $\{0, 1\}$ rather than $\{1, 2\}$). We will make use of succinct representations of S that support the rank and select operators over strings and over binary vectors: $\mathbf{rank}_c(S, i)$ gives the number of occurrences of c in $S[1..i]$ and $\mathbf{select}_c(S, j)$ gives the position in S of the j th occurrence of c .

When $r = 2$, S requires n bits and \mathbf{rank} and \mathbf{select} can be supported in constant time using $\mathcal{O}(n \lg \lg n / \lg n) = o(n)$ bits on top of S [Mun96, Gol06].

Raman et al. [RRR02] devised a bitmap representation that takes $n\mathcal{H}_0(S) + o(n)$ bits, while maintaining the constant time for supporting the operators. For the binary case $\mathcal{H}_0(S)$ is just $m \lg \frac{n}{m} + (n - m) \lg \frac{n}{n-m} = m \lg \frac{n}{m} + \mathcal{O}(m)$, where m is the number of bits set to 1 in S . Golynski et al. [GGG⁺07] reduced the $o(n)$ -bits redundancy in space to $\mathcal{O}(n \lg \lg n / \lg^2 n)$.

When m is much smaller than n , the $o(n)$ -bits term may dominate. Gupta et al. [GHSV06] showed how to achieve space $m \lg \frac{n}{m} + \mathcal{O}(m \lg \lg \frac{n}{m} + \lg n)$ bits, which largely reduces the dependence on n , but now \mathbf{rank} and \mathbf{select} are supported in $\mathcal{O}(\lg m)$ time via binary search [Gup07, Theorem 17 p. 153].

For larger alphabets, of size $r = \mathcal{O}(\text{polylog}(n))$, Ferragina et al. [FMMN07] showed how to represent the sequence within $n\mathcal{H}_0(S) + o(n \lg r)$ bits and support \mathbf{rank} and \mathbf{select} in constant time. Golynski et al. [GRR08, Lemma 9] improved the space to $n\mathcal{H}_0(S) + o(n \lg r / \lg n)$ bits while retaining constant times.

Grossi et al. [GGV03] introduced the so-called *wavelet tree*, which decomposes an arbitrary sequence into several bitmaps. By representing the bitmaps in compressed form [GGG⁺07], the

overall space is $n\mathcal{H}_0(S) + o(n)$ and **rank** and **select** are supported in time $\mathcal{O}(\lg r)$. Multiary wavelet trees decompose the sequence into subsequences over a sublogarithmic-sized alphabet and reduce the time to $\mathcal{O}(1 + \lg r / \lg \lg n)$ [FMMN07, GRR08].

In this article n will generally denote the length of the permutation. All of our $o()$ expressions, even those including several variables, will be asymptotic in n .

2.4 Measures of Presortedness in Permutations

The complexity of *adaptive algorithms*, for problems such as searching, sorting, merging sorted arrays or convex hulls, is studied in the worst case over instances of fixed size *and difficulty*, for a definition of difficulty that is specific to each analysis. Even though sorting a permutation in the comparison model requires $\Theta(n \lg n)$ comparisons in the worst case over permutations of n elements, better results can be achieved for some parameterized classes of permutations. We describe some of those below, see the survey by Moffat and Petersson [MP92] for others.

Knuth [Knu98] considered *runs* (contiguous ascending subsequences) of a permutation π , counted by $\mathbf{nRuns} = 1 + |\{i : 1 \leq i < n, \pi(i+1) < \pi(i)\}|$. Levkopoulos and Petersson [LP94] introduced *Shuffled Up-Sequences* and its generalization *Shuffled Monotone Sequences*, respectively counted by $\mathbf{nSUS} = \min\{k : \pi \text{ is covered by } k \text{ increasing subsequences}\}$, and $\mathbf{nSMS} = \min\{k : \pi \text{ is covered by } k \text{ monotone subsequences}\}$. By definition, $\mathbf{nSMS} \leq \mathbf{nSUS} \leq \mathbf{nRuns}$.

Munro and Spira [MS76] took an orthogonal approach, considering the task of sorting multisets through various algorithms such as MergeSort, showing that they can be adapted to perform in time $\mathcal{O}(n(1 + \mathcal{H}(\langle m_1, \dots, m_r \rangle)))$ where m_i is the number of occurrences of i in the multiset (note this is totally different from our results, that depend on the distribution of the lengths of monotone runs).

Each adaptive sorting algorithm in the comparison model yields a compression scheme for permutations, but the encoding thus defined does not necessarily support the simple application of the permutation to a single element without decompressing the whole permutation, nor the application of the inverse permutation.

3 Contiguous Monotone Runs

Our most fundamental representation takes advantage of permutations that are formed by a few monotone (ascending or descending) runs.

Definition 2 A down step of a permutation π over $[1..n]$ is a position $1 \leq i < n$ such that $\pi(i+1) < \pi(i)$. An ascending run in a permutation π is a maximal range of consecutive positions $[i..j]$ that does not contain any down step. Let d_1, d_2, \dots, d_k be the list of consecutive down steps in π . Then the number of ascending runs of π is noted $\mathbf{nRuns} = k+1$, and the sequence of the lengths of the ascending runs is noted $\mathbf{vRuns} = \langle n_1, n_2, \dots, n_{\mathbf{nRuns}} \rangle$, where $n_1 = d_1, n_2 = d_2 - d_1, \dots, n_{\mathbf{nRuns}-1} = d_k - d_{k-1}$, and $n_{\mathbf{nRuns}} = n - d_k$. (If $k = 0$ then $\mathbf{nRuns} = 1$ and $\mathbf{vRuns} = \langle n \rangle = \langle n \rangle$.) The notions of up step and descending run are defined similarly.

For example, the permutation $(1, 3, 5, 7, 9, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10})$ contains $\mathbf{nRuns} = 2$ ascending runs, of lengths forming the vector $\mathbf{vRuns} = \langle 5, 5 \rangle$.

We now describe a data structure that represents a permutation partitioned into \mathbf{nRuns} ascending runs, and is able to compute any $\pi(i)$ and $\pi^{-1}(i)$.

3.1 Structure

Construction We find the down-steps of π in linear time, obtaining **nRuns** runs of lengths **vRuns** = $\langle n_1, \dots, n_{\text{nRuns}} \rangle$, and then apply the Huffman algorithm to the vector **vRuns**. When we set up the leaves v of the Huffman tree, we store their original index in **vRuns**, $idx(v)$, and the starting position in π of their corresponding run, $pos(v)$. After the tree is built, we use $idx(v)$ to compute a permutation ϕ over $[1..\text{nRuns}]$ so that $\phi(i) = j$ if the leaf corresponding to n_i is placed at the j th left-to-right leaf in the Huffman tree. We also compute ϕ^{-1} . We also precompute a bitmap $C[1..n]$ that marks the beginning of runs in π and give constant-time support for **rank** and **select**. Since C contains only **nRuns** bits set out of n , it is represented in compressed form [GGG⁺07] within $\text{nRuns} \lg \frac{n}{\text{nRuns}} + o(n)$ bits.

Now we set a new permutation π' over $[1..n]$ where the runs are written in the order given by ϕ^{-1} : We first copy from π the run whose endpoints are those of the leftmost tree leaf, then the run pointed by the second leftmost leaf, and so on. Simultaneously, we compute $pos'(v)$ for the leaves v , denoting the starting position of the area they cover in π' . After creating π' the original permutation π can be deleted. We say that an internal node *covers* the contiguous area of π' formed by concatenating the runs of all the leaves that descend from v . We compute, for all nodes v , $pos'(v)$, the starting position of the area covered by v in π' , $length(v)$, the size of that area, and $leaves(v)$, the number of leaves that descend from v .

Now we enhance the Huffman tree into a wavelet-tree-like structure [GGV03] without altering its shape, as follows. Starting from the root, first process recursively each child. For the leaves we do nothing. Once the left and right children, v_l and v_r , of an internal node v have been processed, the invariant is that the areas they cover have already been sorted. We create a bitmap for v , of size $length(v)$. Now we merge the areas of v_l and v_r in time $\mathcal{O}(length(v))$. As we do the merging, each time we take an element from v_l we append a bit 0 to the node bitmap, and a bit 1 when we take an element from v_r . When we finish, π' has been sorted and we can delete it. The Huffman-shaped wavelet tree (only with fields *leaves* and *pos*), ϕ , and C represent π .

Space and construction cost Note that each of the n_i elements of leaf i (at depth ℓ_i) is merged ℓ_i times, contributing ℓ_i bits to the bitmaps of its ancestors, and thus the total number of bits in all bitmaps is $\sum n_i \ell_i$. Thus the total number of bits in the Huffman-shaped wavelet tree is at most $n(1 + \mathcal{H}(\text{vRuns}))$. Those bitmaps, however, are represented in compressed form [GGG⁺07], which allows us removing the n extra bits added by the Huffman encoding.

Let us call $m_j = n_{\phi^{-1}(j)}$ the length of the run corresponding to the j th left-to-right leaf, and $m_{i,j} = m_i + \dots + m_j$. The compressed representation [GGG⁺07] takes, on a bitmap of length n and m 1s, $m \lg \frac{n}{m} + (n-m) \lg \frac{n}{n-m}$ bits, plus a redundancy of $\mathcal{O}(n \lg \lg n / \lg^2 n)$ bits. We prove by induction (see also Grossi et al. [GGV03]) that the compressed space allocated for all the bitmaps descending from a node covering leaves $[i..k]$ is $\sum_{i \leq r \leq k} m_r \lg \frac{m_{i,k}}{m_r}$ (we consider the redundancy later). Consider two sibling leaves merging two runs of m_i and m_{i+1} elements. Their parent bitmap contains m_i 0s and m_{i+1} 1s, and thus its compressed representation requires $m_i \lg \frac{m_i + m_{i+1}}{m_i} + m_{i+1} \lg \frac{m_i + m_{i+1}}{m_{i+1}}$ bits. Now consider a general Huffman tree node merging a left subtree covering leaves $[i..j]$ and a right subtree covering leaves $[j+1..k]$. Then the bitmap of the node will be compressed to $m_{i,j} \lg \frac{m_{i,k}}{m_{i,j}} + m_{j+1,k} \lg \frac{m_{i,k}}{m_{j+1,k}}$ bits. By the inductive hypothesis, all the bitmaps on the left child and its subtrees add up to $\sum_{i \leq r \leq j} m_r \lg \frac{m_{i,j}}{m_r}$, and those on the right add up to $\sum_{j+1 \leq r \leq k} m_r \lg \frac{m_{j+1,k}}{m_r}$. Adding up the three formulas we get the inductive thesis.

Therefore, a compressed representation of the bitmaps requires $n\mathcal{H}(\mathbf{vRuns})$ bits, plus the redundancy. The latter, added over all the bitmaps, is $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vRuns})) \lg \lg n / \lg^2 n) = o(n)$ because $\mathcal{H}(\mathbf{vRuns}) \leq \lg n$.¹ To this we must add the $\mathcal{O}(\mathbf{nRuns} \lg n)$ bits of the tree pointers and extra data like *pos* and *leaves*, the $\mathcal{O}(\mathbf{nRuns} \lg \mathbf{nRuns})$ bits for ϕ , and the $\mathbf{nRuns} \lg \frac{n}{\mathbf{nRuns}} + o(n)$ bits for C .

The construction time is $\mathcal{O}(\mathbf{nRuns} \lg \mathbf{nRuns})$ for the Huffman algorithm, plus $\mathcal{O}(\mathbf{nRuns})$ for computing ϕ and filling the node fields like *pos* and *leaves*, plus $\mathcal{O}(n)$ for constructing π' and C , plus the total number of bits appended to all bitmaps, which includes the merging cost. The extra structures for **rank** are built in linear time on those bitmaps.² All this adds up to $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vRuns})))$, because $\mathbf{nRuns} \lg \mathbf{nRuns} \leq n\mathcal{H}(\mathbf{vRuns}) + \lg n$ by concavity, recall Definition 1.

3.2 Queries

Computing π and π^{-1} One can regard the wavelet tree as a device that tracks the evolution of a merge-sorting of π' , so that in the bottom we have (conceptually) the sequence π' (with one run per leaf) and in the top we have (conceptually) the sorted permutation $(1, 2, \dots, n)$.

To compute $\pi^{-1}(j)$ we start at the top and find out where that position came from in π' . We start at offset $j' = j$ of the root bitmap B . If $B[j'] = 0$, then position j' came from the left subtree in the merging. Thus we go down to the left child with $j' \leftarrow \mathbf{rank}_0(B, j')$, which is the position of j' in the array of the left child before the merging. Otherwise we go down to the right child with $j' \leftarrow \mathbf{rank}_1(B, j')$. We continue recursively until we reach a leaf v . At this point we know that j came from the corresponding run, at offset j' , that is, $\pi^{-1}(j) = \mathbf{pos}(v) + j' - 1$.

To compute $\pi(i)$ we do the reverse process, but we must first determine the leaf v and offset i' within v corresponding to position i : We compute $l = \phi(\mathbf{rank}_1(C, i))$, so that i falls at the l th left-to-right leaf. Then we traverse the Huffman tree down so as to find the l th leaf. This is easily done as we have *leaves*(v) stored at internal nodes. Upon arriving at leaf v , we know that the offset is $i' = i - \mathbf{pos}(v) + 1$. We now start an upward traversal from v using the nodes that are already in the recursion stack. If v is a left child of its parent u , then we set $i' \leftarrow \mathbf{select}_0(B, i')$ to locate it in the merged array of the parent, else we set $i' \leftarrow \mathbf{select}_1(B, i')$, where B is the bitmap of u . Then we set $v \leftarrow u$ and continue until reaching the root, where we answer $\pi(i) = i'$.

Query time In both queries the time is $\mathcal{O}(\ell)$, where ℓ is the depth of the leaf arrived at. If i is chosen uniformly at random in $[1..n]$, then the average cost is $\frac{1}{n} \sum n_i \ell_i = \mathcal{O}(1 + \mathcal{H}(\mathbf{vRuns}))$. However, the worst case can be $\mathcal{O}(\mathbf{nRuns})$ in a fully skewed tree. We can ensure $\ell = \mathcal{O}(\lg \mathbf{nRuns})$ in the worst case while maintaining the average case by slightly rebalancing the Huffman tree [ML01]. Given any constant $x > 0$, the height of the Huffman tree can be bound to at most $(1 + x) \lg \mathbf{nRuns}$ so that the total number of bits added to the encoding is at most $n \cdot \mathbf{nRuns}^{-x \lg \varphi}$, where $\varphi \approx 1.618$ is the golden ratio. This is $o(n)$ if $\mathbf{nRuns} = \omega(1)$, and otherwise the cost was $\mathcal{O}(\mathbf{nRuns}) = \mathcal{O}(1)$ anyway. Similarly, the average time stays $\mathcal{O}(1 + \mathcal{H}(\mathbf{vRuns}))$, as it increases at most by $\mathcal{O}(\mathbf{nRuns}^{-x \lg \varphi}) = \mathcal{O}(1)$. This rebalancing takes just $\mathcal{O}(\mathbf{nRuns})$ time if the frequencies are already sorted.

Note also that the space required by the query is $\mathcal{O}(\lg \mathbf{nRuns})$. This can be made constant by storing parent pointers in the wavelet tree, which does not change the asymptotic space.

¹To make sure this is $o(n)$ even if there are many short bitmaps, we can concatenate all the bitmaps into a single one, and replace pointers to bitmaps by offsets to this single bitmap. Operations **rank** and **select** translate easily into a concatenated bitmap.

²While the linear construction time is not obvious from their article [GGG⁺07], a subsequent result [P08] achieved even less redundancy and linear construction time.

Theorem 1 *There is an encoding scheme using at most $n\mathcal{H}(\mathbf{vRuns}) + \mathcal{O}(\mathbf{nRuns} \lg n) + o(n)$ bits to represent a permutation π over $[1..n]$ covered by \mathbf{nRuns} contiguous ascending runs of lengths forming the vector \mathbf{vRuns} . It can be built within time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vRuns})))$, and supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1 + \lg \mathbf{nRuns})$ for any value of $i \in [1..n]$. If i is chosen uniformly at random in $[1..n]$ then the average computation time is $\mathcal{O}(1 + \mathcal{H}(\mathbf{vRuns}))$.*

We note that the space analysis leading to $n\mathcal{H}(\mathbf{vRuns}) + o(n)$ bits works for any tree shape. We could have used a balanced tree, yet we would not achieve $\mathcal{O}(1 + \mathcal{H}(\mathbf{vRuns}))$ average time. On the other hand, by using Hu-Tucker codes instead of Huffman, as in our previous work [BN09], we would not need the permutation ϕ and, by using compact tree representations [SN10], we would be able to reduce the space to $n\mathcal{H}(\mathbf{vRuns}) + \mathcal{O}(\mathbf{nRuns} \lg \frac{n}{\mathbf{nRuns}}) + o(n)$. This is interesting for large values of \mathbf{nRuns} , as it is always $n\mathcal{H}(\mathbf{vRuns}) + o(n(1 + \mathcal{H}(\mathbf{vRuns})))$ even if $\mathbf{nRuns} = \Theta(n)$.³

3.3 Mixing Ascending and Descending Runs

We can easily extend Theorem 1 to mix ascending and descending runs.

Corollary 2 *Theorem 1 holds verbatim if π is partitioned into a sequence \mathbf{nRuns} contiguous monotone (i.e., ascending or descending) runs of lengths forming the vector \mathbf{vRuns} .*

Proof. We mark in a bitmap of length \mathbf{nRuns} whether each run is ascending or descending, and then reverse descending runs in π , so as to obtain a new permutation π_{asc} , which is represented using Theorem 1 (some runs of π could now be merged in π_{asc} , but this only reduces $\mathcal{H}(\mathbf{vRuns})$, recall Definition 1).

The values $\pi(i)$ and $\pi^{-1}(j)$ are easily computed from π_{asc} : If $\pi_{asc}^{-1}(j) = i$, we use C to determine that i is within run $\pi_{asc}(\ell..r)$, that is, $\ell = \text{select}_1(\text{rank}_1(C, i))$ and $r = \text{select}_1(\text{rank}_1(C, i) + 1) - 1$. If that run is reversed in π , then $\pi^{-1}(j) = \ell + r - i$, else $\pi^{-1}(j) = i$. For $\pi(i)$, we use C to determine that i belongs to run $\pi(\ell..r)$. If the run is descending, then we return $\pi_{asc}(\ell + r - i)$, else we return $\pi_{asc}(i)$. The operations on C require only constant time. The extra construction time is just $\mathcal{O}(n)$, and no extra space is needed apart from $\mathbf{nRuns} = o(\mathbf{nRuns} \lg n)$ bits. \square

Note that, unlike the case of ascending runs, where there is an obviously optimal way of partitioning (that is, maximize the run lengths), we have some freedom when partitioning into ascending or descending runs, at the endpoints of the runs: If an ascending (resp. descending) run is followed by a descending (resp. ascending) run, the limiting element can be moved to either run; if two ascending (resp. descending) runs are consecutive, one can create a new descending (resp. ascending) run with the two endpoint elements. While finding the optimal partitioning might not be easy, we note that these decisions cannot affect more than $\mathcal{O}(\mathbf{nRuns})$ elements, and thus the entropy of the partition cannot be modified by more than $\mathcal{O}(\mathbf{nRuns} \lg n)$, which is absorbed by the redundancy of our representation.

³We do not follow this path because we are more interested in multiary codes (see Section 3.5) and, to the best of our knowledge, there is no efficient (i.e., $\mathcal{O}(\mathbf{nRuns} \lg \mathbf{nRuns})$ time) algorithm for building multiary Hu-Tucker codes [Knu98].

3.4 Improved Adaptive Sorting

One of the best known sorting algorithms is MergeSort, based on a simple linear procedure to merge two already sorted arrays, and with a worst case complexity of $n \lceil \lg n \rceil$ comparisons and $\mathcal{O}(n \lg n)$ running time. It had been already noted [Knu98] that finding the down-steps of the array in linear time allows improving the time of MergeSort to $\mathcal{O}(n(1 + \lg \mathbf{nRuns}))$ (the down-step concept can be applied to general sequences, where consecutive equal values do not break runs).

We now show that the construction process of our data structure sorts the permutation and, applied on a general sequence, it achieves a refined sorting time of $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vRuns}))) \subset \mathcal{O}(n(1 + \lg \mathbf{nRuns}))$ (since $\mathcal{H}(\mathbf{vRuns}) \leq \lg \mathbf{nRuns}$).

Theorem 3 *There is an algorithm sorting an array of length n covered by \mathbf{nRuns} contiguous monotone runs of lengths forming the vector \mathbf{vRuns} in time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vRuns})))$, which is worst-case optimal in the comparison model.*

Proof. Our wavelet tree construction of Theorem 1 (and Corollary 2) indeed sorts π within this time, and it also works if the array is not a permutation. This is optimal because, even considering just ascending runs, there are $\frac{n!}{n_1!n_2!\dots n_{\mathbf{nRuns}}!}$ different permutations that can be covered with runs of lengths forming the vector $\mathbf{vRuns} = \langle n_1, n_2, \dots, n_{\mathbf{nRuns}} \rangle$. Thus $\lg \frac{n!}{n_1!n_2!\dots n_{\mathbf{nRuns}}!}$ comparisons are necessary. Using Stirling's approximation to the factorial we have $\lg \frac{n!}{n_1!n_2!\dots n_{\mathbf{nRuns}}!} = (n + 1/2) \lg n - \sum_i (n_i + 1/2) \lg n_i - \mathcal{O}(\lg \mathbf{nRuns})$. Since $\sum \lg n_i \leq \mathbf{nRuns} \lg(n/\mathbf{nRuns})$, this is $n\mathcal{H}(\mathbf{vRuns}) - \mathcal{O}(\mathbf{nRuns} \lg(n/\mathbf{nRuns})) = n\mathcal{H}(\mathbf{vRuns}) - \mathcal{O}(n)$. The term $\Omega(n)$ is also necessary to read the input, hence implying a lower bound of $\Omega(n(1 + \mathcal{H}(\mathbf{vRuns})))$.

Note, however, that the set of permutations that *can be* covered with \mathbf{nRuns} runs of lengths \mathbf{vRuns} , may contain permutations that can be covered with fewer runs (as two consecutive runs could be merged), and thus they have entropy less than $\mathcal{H}(\mathbf{vRuns})$, recall Definition 1. We have proved that the lower bound applies to the union of two classes: one (1) contains (some⁴) permutations of entropy $\mathcal{H}(\mathbf{vRuns})$ and the other (2) contains (some) permutations of entropy less than $\mathcal{H}(\mathbf{vRuns})$. Obviously the bound does not hold for class (2) alone, as we can sort it in less time. Since we can tell the class of a permutation in $\mathcal{O}(n)$ time by counting the down-steps, it follows that the bound also applies to class (1) alone (otherwise $\mathcal{O}(n) + o(n\mathcal{H}(\mathbf{vRuns}))$ would be achievable for (1)+(2)). \square

3.5 Boosting Time Performance

The time performance achieved in Theorem 1 (and Corollary 2) can be boosted by an $\mathcal{O}(\lg \lg n)$ time factor by using Huffman codes of higher arity.

Given the run lengths \mathbf{vRuns} , we build the t -ary Huffman tree for \mathbf{vRuns} , with $t = \sqrt{\lg n}$. Since now we merge t children to build the parent, the sequence stored in the parent to indicate the child each element comes from is not binary, but over alphabet $[1..t]$. In addition, we set up \mathbf{nRuns} pointers to provide direct access to the leaves, and parent pointers.

The total length of all the sequences stored at all the Huffman tree nodes is $< n(1 + \mathcal{H}(\mathbf{vRuns})/\lg t)$ [Huf52]. To reduce the redundancy, we represent each sequence $S[1..m]$ stored

⁴Other permutations with vectors distinct from \mathbf{vRuns} could also have entropy $\mathcal{H}(\mathbf{vRuns})$.

at a node using the compressed representation of Golynski et al. [GRR08, Lemma 9], which yields space $m\mathcal{H}_0(S) + \mathcal{O}(m \lg t \lg \lg m / \lg^2 m)$ bits.

For the string $S[1..m]$ corresponding to a leaf covering run lengths m_1, \dots, m_t , we have $m\mathcal{H}_0(S) = \sum m_i \lg \frac{m}{m_i}$. From there we can carry out exactly the same analysis done in Section 3.1 for binary trees, to conclude that the sum of the $m\mathcal{H}_0(S)$ bits for all the strings S over all the tree nodes is $n\mathcal{H}(\mathbf{vRuns})$. On the other hand, the redundancies add up to $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vRuns})/\lg t) \lg t \lg \lg n / \lg^2 n) = o(n)$ bits.⁵

The advantage of the t -ary representation is that the average leaf depth is $1 + \mathcal{H}(\mathbf{vRuns})/\lg t = \mathcal{O}(1 + \mathcal{H}(\mathbf{vRuns})/\lg \lg n)$. The algorithms to compute $\pi(i)$ and $\pi^{-1}(i)$ are similar, except that **rank** and **select** are carried out on sequences S over alphabets of size $\sqrt{\lg n}$. Those operations can still be carried out in constant time on the representation we have chosen [GRR08]. The only detail is that, for $\pi(i)$ we first moved from the root to the leaf using the field $leaves(v)$. This does not anymore allow us processing a node in constant time, and thus we have opted for storing an array of pointers to the leaves and parent pointers.

For the worst case, if $\mathbf{nRuns} = \omega(1)$, we can again limit the depth of the Huffman tree to $\mathcal{O}(\lg \mathbf{nRuns} / \lg \lg n)$ and maintain the same average time. The multiary case is far less understood than the binary case. Recently, an algorithm to find the optimal length-restricted t -ary code has been presented whose running time is linear once the lengths are sorted [Bae07]. To analyze the increase in redundancy, consider the sub-optimal method that simply takes any node v of depth more than $\ell = 4 \lg \mathbf{nRuns} / \lg t$ and balances its subtree (so that height $5 \lg \mathbf{nRuns} / \lg t$ is guaranteed). Since any node at depth ℓ covers a total length of at most $n/t^{\lfloor \ell/2 \rfloor}$ (see next paragraph), the sum of all the lengths covered by these nodes is at most $\mathbf{nRuns} \cdot n/t^{\lfloor \ell/2 \rfloor}$. By forcing those subtrees to be balanced, the average leaf depth increases by at most $(\lg \mathbf{nRuns} / \lg t) \mathbf{nRuns} / t^{\lfloor \ell/2 \rfloor} \leq \lg(\mathbf{nRuns}) / (\mathbf{nRuns} \lg t) = \mathcal{O}(1)$. Hence the worst case is limited to $\mathcal{O}(1 + \lg \mathbf{nRuns} / \lg \lg n)$ while the average case stays within $\mathcal{O}(1 + \mathcal{H}(\mathbf{vRuns})/\lg \lg n)$. For the space we need a finer consideration: As $\mathbf{nRuns} = \omega(1)$, the increase in average leaf depth is $o(1/\lg t)$. Since increasing by one the depth of a leaf covering m elements costs $m \lg t$ further bits, the total increase in space redundancy is $o(n)$.

The limit on the probability is obtained as follows. Consider a node v in the t -ary Huffman tree. Then $length(u) \geq length(v)$ for any uncle u of v , as otherwise switching v and u improves the already optimal Huffman tree. Hence w , the grandparent of v (i.e., the parent of u) must cover an area of size $length(w) \geq t \cdot length(v)$. Thus the covered length is multiplied at least by t when moving from a node to its grandparent. Conversely, it is divided at least by t as we move from a node to any grandchild. As the total length at the root is n , the length covered by any node v at depth ℓ is at most $length(v) \leq n/t^{\lfloor \ell/2 \rfloor}$.

This yields our final result for contiguous monotone runs.

Theorem 4 *There is an encoding scheme using at most $n\mathcal{H}(\mathbf{vRuns}) + \mathcal{O}(\mathbf{nRuns} \lg n) + o(n)$ bits to encode a permutation π over $[1..n]$ covered by \mathbf{nRuns} contiguous monotone runs of lengths forming the vector \mathbf{vRuns} . It can be built within time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vRuns})/\lg \lg n))$, and supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1 + \lg \mathbf{nRuns} / \lg \lg n)$ for any value of $i \in [1..n]$. If i is chosen uniformly at random in $[1..n]$ then the average computation time is $\mathcal{O}(1 + \mathcal{H}(\mathbf{vRuns})/\lg \lg n)$.*

The only missing part is the construction time, since now we have to build strings $S[1..m]$ by merging t increasing runs. This can be done in $\mathcal{O}(m)$ time by using atomic heaps [FW94]. The compressed sequence representations are built in linear time [GRR08]. Note this implies that we

⁵Again, we can concatenate all the sequences to make sure this redundancy is asymptotic in n .

can sort an array with **nRuns** contiguous monotone runs of lengths forming the vector **vRuns** in time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vRuns})/\lg \lg n))$, yet we are not anymore within the comparison model.

3.6 An Improved Sequence Representation

Interestingly, the previous result yields almost directly a new representation of sequences that, compared to the state of the art [FMMN07, GRR08], provides improved average time performance.

Theorem 5 *Given a string $S[1..n]$ over alphabet $[1..\sigma]$ with zero-order entropy $\mathcal{H}_0(S)$, there is an encoding for S using at most $n\mathcal{H}_0(S) + \mathcal{O}(\sigma \lg n) + o(n)$ bits and answering queries $S[i]$, **rank** _{c} (S, i) and **select** _{c} (S, i) in time $\mathcal{O}(1 + \lg \sigma / \lg \lg n)$ for any $c \in [1..\sigma]$ and $i \in [1..n]$. When i is chosen at random in query $S[i]$, or c is chosen with probability n_c/n in queries **rank** _{c} (S, i) and **select** _{c} (S, i), where n_c is the frequency of c in S , the average query time is $\mathcal{O}(1 + \mathcal{H}_0(S)/\lg \lg n)$.*

Proof. We build exactly the same t -ary Huffman tree used in Theorem 4, using the frequencies n_c instead of run lengths. The sequences at each internal node are formed so as to indicate how the symbols in the child nodes are interleaved in S . This is precisely a multiary Huffman-shaped wavelet tree [GGV03, FMMN07], and our previous analysis shows that the space used by the tree is exactly as in Theorem 4, where now the entropy is $\mathcal{H}_0(S) = \sum_c \frac{n_c}{n} \lg \frac{n}{n_c}$. The three queries are solved by going down or up the tree and using **rank** and **select** on the sequences stored at the nodes [GGV03, FMMN07]. Under the conditions stated for the average case, one arrives at the leaf of symbol c with probability n_c/n , and then the average case complexities follow. \square

4 Strict Runs

Some classes of permutations can be covered by a small number of runs of a stricter type. We present an encoding scheme that take advantage of them.

Definition 3 *A strict ascending run in a permutation π is a maximal range of positions satisfying $\pi(i+k) = \pi(i) + k$. The head of such run is its first position. The number of strict ascending runs of π is noted **nSRuns**, and the sequence of the lengths of the strict ascending runs is noted **vSRuns**. We will call **vHRuns** the sequence of contiguous monotone run lengths of the sequence formed by the strict run heads of π . Similarly, the notion of a strict descending run can be defined, as well as that of strict (monotone) run encompassing both.*

For example, the permutation $(6, 7, 8, 9, 10, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5})$ contains **nSRuns** = 2 strict runs, of lengths **vSRuns** = $\langle 5, 5 \rangle$. The run heads are $\langle 6, \mathbf{1} \rangle$, which form 1 monotone run, of lengths **vHRuns** = $\langle 2 \rangle$. Instead, the permutation $(1, 3, 5, 7, 9, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10})$ contains **nSRuns** = 10 strict runs, each of length 1.

Theorem 6 *Assume there is an encoding P for a permutation over $[1..n]$ with **nRuns** contiguous monotone runs of lengths forming the vector **vRuns**, which requires $s(n, \mathbf{nRuns}, \mathbf{vRuns})$ bits of space and can apply the permutation and its inverse in time $t(n, \mathbf{nRuns}, \mathbf{vRuns})$. Now consider a permutation π over $[1..n]$ covered by **nSRuns** strict runs and by **nRuns** \leq **nSRuns** monotone runs, and let **vHRuns** be the vector formed by the **nRuns** monotone run lengths in the permutation of strict run heads. Then there is an encoding scheme using at most $s(\mathbf{nSRuns}, \mathbf{nRuns}, \mathbf{vHRuns}) +$*

$\mathcal{O}(\text{nSRuns} \lg \frac{n}{\text{nSRuns}}) + o(n)$ bits for π . It can be computed in $\mathcal{O}(n)$ time on top of that for building P . It supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(t(\text{nSRuns}, \text{nRuns}, \text{vHRuns}))$ for any value $i \in [1..n]$.

Proof. We first set up a bitmap R of length n marking with a 1 bit the beginning of the strict runs. We set up a second bitmap R^{inv} such that $R^{inv}[i] = R[\pi^{-1}(i)]$. Now we create a new permutation π' over $[1..\text{nSRuns}]$ which collapses the strict runs of π , $\pi'(i) = \text{rank}_1(R^{inv}, \pi(\text{select}_1(R, i)))$. All this takes $\mathcal{O}(n)$ time and the bitmaps take $2\text{nSRuns} \lg \frac{n}{\text{nSRuns}} + \mathcal{O}(\text{nSRuns}) + o(n)$ bits in compressed form [GGG⁺07], where **rank** and **select** are supported in constant time.

Now we build the structure P for π' . The number of monotone runs in π is the same as for the sequence of strict run heads in π , and in turn the same as the runs in π' . So the number of runs in π' is also nRuns and their lengths are vHRuns . Thus we require $s(\text{nSRuns}, \text{nRuns}, \text{vHRuns})$ further bits.

To compute $\pi(i)$, we find $i' \leftarrow \text{rank}_1(R, i)$ and then compute $j' \leftarrow \pi'(i')$. The final answer is $\text{select}_1(R^{inv}, j') + i - \text{select}_1(R, i')$. To compute $\pi^{-1}(j)$, we find $j' \leftarrow \text{rank}_1(R^{inv}, j)$ and then compute $i' \leftarrow (\pi')^{-1}(j')$. The final answer is $\text{select}_1(R, i') + j - \text{select}_1(R^{inv}, j')$. The structure requires only constant time on top of that to support the operator $\pi'()$ and its inverse $\pi'^{-1}()$. \square

The theorem can be combined with previous results, for example Theorem 4, in order to obtain concrete data structures. This representation is interesting because its space could be much less than n if nSRuns is small enough. However, it still retains an $o(n)$ term that can be dominant. The following corollary describes a compressed data structure where the $o(n)$ term is significantly reduced.

Corollary 7 *The $o(n)$ term in the space of Theorem 6 can be replaced by $\mathcal{O}(\text{nSRuns} \lg \lg \frac{n}{\text{nSRuns}} + \lg n)$ at the cost of $\mathcal{O}(1 + \lg \text{nSRuns})$ extra time for the queries.*

Proof. Replace the structure of Golynski et al. [GGG⁺07] by the binary searchable gap encoding of Gupta et al. [GHSV06], which takes $\mathcal{O}(1 + \lg \text{nSRuns})$ time for **rank** and **select** (recall Section 2.3). \square

Other tradeoffs for the bitmap encodings are possible, such as the one described by Gupta [Gup07, Theorem 18 p. 155].

5 Shuffled Sequences

Up to now our runs have been contiguous in π . Levkopoulos and Petersson [LP94] introduced the more sophisticated concept of partitions formed by interleaved runs, such as *Shuffled UpSequences* (SUS) and *Shuffled Monotone Sequences* (SMS). We now show how to take advantage of permutations formed by shuffling (interleaving) a small number of runs.

Definition 4 *A decomposition of a permutation π over $[1..n]$ into Shuffled UpSequences is a set of, not necessarily consecutive, subsequences of increasing numbers that have to be removed from π in order to reduce it to the empty sequence. The number of shuffled upsequences in such a decomposition of π is noted nSUS , and the vector formed by the lengths of the involved shuffled upsequences, in arbitrary order, is noted vSUS . When the subsequences can be of increasing or decreasing numbers,*

we call them Shuffled Monotone Sequences, call \mathbf{nSMS} their number and \mathbf{vSMS} the vector formed by their lengths.

For example, the permutation $(1, 6, 2, 7, 3, 8, 4, 9, 5, 10)$ contains $\mathbf{nSUS} = 2$ shuffled upsequences of lengths forming the vector $\mathbf{vSUS} = \langle 5, 5 \rangle$, but $\mathbf{nRuns} = 5$ runs, all of length 2. Interestingly, we can reduce the problem of representing shuffled sequences to that of representing strings and contiguous runs.

5.1 Reduction to Strings and Contiguous Monotone Sequences

We first show how a permutation with a small number of shuffled monotone sequences can be represented using strings over a small alphabet and permutations with a small number of contiguous monotone sequences.

Theorem 8 *Assume there exists an encoding P for a permutation over $[1..n]$ with \mathbf{nRuns} contiguous monotone runs of lengths forming the vector \mathbf{vRuns} , which requires $s(n, \mathbf{nRuns}, \mathbf{vRuns})$ bits of space and supports the application of the permutation and its inverse in time $t(n, \mathbf{nRuns}, \mathbf{vRuns})$. Assume also that there is a data structure S for a string $S[1..n]$ over an alphabet of size \mathbf{nSMS} with symbol frequencies \mathbf{vSMS} , using $s'(n, \mathbf{nSMS}, \mathbf{vSMS})$ bits of space and supporting operators **rank**, **select**, and access to values $S[i]$, in time $t'(n, \mathbf{nSMS}, \mathbf{vSMS})$. Now consider a permutation π over $[1..n]$ covered by \mathbf{nSMS} shuffled monotone sequences of lengths \mathbf{vSMS} . Then there exists an encoding of π using at most $s(n, \mathbf{nSMS}, \mathbf{vSMS}) + s'(n, \mathbf{nSMS}, \mathbf{vSMS}) + \mathcal{O}(\mathbf{nSMS} \lg \frac{n}{\mathbf{nSMS}}) + o(n)$ bits. Given the covering into SMSs, the encoding can be built in time $\mathcal{O}(n)$, in addition to that of building P and S . It supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $t(n, \mathbf{nSMS}, \mathbf{vSMS}) + t'(n, \mathbf{nSMS}, \mathbf{vSMS})$ for any value of $i \in [1..n]$. The result is also valid for shuffled upsequences, in which case P is just required to handle ascending runs.*

Proof. Given the partition of π into \mathbf{nSMS} monotone subsequences, we create a string $S[1..n]$ over alphabet $[1..\mathbf{nSMS}]$ that indicates, for each element of π , the label of the monotone sequence it belongs to. We encode $S[1..n]$ using the data structure S . We also store an array $A[1..\mathbf{nSMS}]$ so that $A[\ell]$ is the accumulated length of all the sequences with label less than ℓ .

Now consider the permutation π' formed by the sequences taken in label order: π' can be covered with \mathbf{nSMS} contiguous monotone runs \mathbf{vSMS} , and hence can be encoded using $s(n, \mathbf{nSMS}, \mathbf{vSMS})$ additional bits using P . This supports the operators $\pi'()$ and $\pi'^{-1}()$ in time $t(n, \mathbf{nSMS}, \mathbf{vSMS})$ (again, some of the runs could be merged in π' , which only improves time and space in P). Thus $\pi(i) = \pi'(A[S[i]] + \mathbf{rank}_{S[i]}(S, i))$ can be computed in time $t(n, \mathbf{nSMS}, \mathbf{vSMS}) + t'(n, \mathbf{nSMS}, \mathbf{vSMS})$. Similarly, $\pi^{-1}(i) = \mathbf{select}_\ell(S, (\pi')^{-1}(i) - A[\ell])$, where ℓ is such that $A[\ell] < (\pi')^{-1}(i) \leq A[\ell + 1]$, can also be computed in time $t(n, \mathbf{nSMS}, \mathbf{vSMS}) + t'(n, \mathbf{nSMS}, \mathbf{vSMS})$, plus the time to find ℓ . The latter is reduced to constant by representing A with a bitmap $A'[1..n]$ with the bits set at the values $A[\ell] + 1$, so that $A[\ell] = \mathbf{select}_1(A', \ell) - 1$, and the binary search is replaced by $\ell = \mathbf{rank}_1(A', (\pi')^{-1}(i))$. With the structure of Golynski et al. [GGG⁺07], A' uses $\mathcal{O}(\mathbf{nSMS} \lg \frac{n}{\mathbf{nSMS}}) + o(n)$ bits and operates in constant time. \square

We will now obtain concrete results by using specific representations for P and S , and specific methods to find the decomposition into shuffled sequences.

5.2 Shuffled UpSequences

Given an arbitrary permutation, one can decompose it in linear time into contiguous runs in order to minimize $\mathcal{H}(\mathbf{vRuns})$, where \mathbf{vRuns} is the vector of run lengths. However, decomposing the same permutation into shuffled up (resp. monotone) sequences so as to minimize either \mathbf{nSUS} or $\mathcal{H}(\mathbf{vSUS})$ (resp. \mathbf{nSMS} or $\mathcal{H}(\mathbf{vSMS})$) is computationally harder.

Fredman [Fre75] gave an algorithm to compute a partition of minimum size \mathbf{nSUS} , into upsequences, claiming a worst case complexity of $\mathcal{O}(n \lg n)$. Even though he did not claim it at the time, it is easy to observe that his algorithm is adaptive in \mathbf{nSUS} and takes $\mathcal{O}(n(1 + \lg \mathbf{nSUS}))$ time. We give here an improvement of his algorithm that computes the partition itself within time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vSUS})))$, no worse than the time of his original algorithm, as $\mathcal{H}(\mathbf{vSUS}) \leq \lg \mathbf{nSUS}$.

Theorem 9 *If an array $D[1..n]$ can be optimally covered by \mathbf{nSUS} shuffled upsequences (equal values do not break an upsequence), then there is an algorithm finding a covering of size \mathbf{nSUS} in time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vSUS}))) \subset \mathcal{O}(n(1 + \lg \mathbf{nSUS}))$, where \mathbf{vSUS} is the vector formed by the lengths of the upsequences found.*

Proof. Initialize a sequence $S_1 = (D[1])$, and a splay tree T [ST85] with the node (S_1) , ordered by the rightmost value of the sequence contained by each node. For each further array element $D[i]$, search for the sequence with the maximum ending point no larger than $D[i]$. If it exists, add $D[i]$ to this sequence, otherwise create a new sequence and add it to T .

Fredman [Fre75] already proved that this algorithm finds a partition of minimum size \mathbf{nSUS} . Note that, although the rightmost values of the splay tree nodes change when we insert a new element in their sequence, their relative position with respect to the other nodes remains the same, since all the nodes at the right hold larger values than the one inserted. This implies in particular that only searches and insertions are performed in the splay tree.

A simple analysis, valid for both the plain sorted array in Fredman's proof and the splay tree of our own proof, yields an adaptive complexity of $\mathcal{O}(n(1 + \lg \mathbf{nSUS}))$ comparisons, since both structures contain at most \mathbf{nSUS} elements at any time. The additional linear term (relevant when $\mathbf{nSUS} = 1$) corresponds to the cost of reading each element once.

The analysis of the algorithm using the splay tree refines the complexity to $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vSUS})))$, where \mathbf{vSUS} is the vector formed by the lengths of the upsequences found. These lengths correspond to the frequencies of access to each node of the splay tree, which yields the total access time of $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vSUS})))$ [ST85, Theorem 2]. \square

The theorem obviously applies to the particular case where the array is a permutation. For permutations and, in general, integer arrays over a universe $[1..m]$, we can deviate from the comparison model and find the partition within time $\mathcal{O}(n \lg \lg m)$, by using y -fast tries [Wil83] instead of splay trees.

We can now give a concrete representation for shuffled upsequences. The complete description of the permutation requires to encode the computation the partitioning and of the comparisons performed by the sorting algorithm. This time the encoding cost of partitioning is as important as that of merging.

Theorem 10 *Let π be a permutation over $[1..n]$ that can be optimally covered by \mathbf{nSUS} shuffled upsequences, and let \mathbf{vSUS} be the vector formed by the lengths of the decomposition found by the algorithm of Theorem 9. Then there is an encoding scheme for π using at most $2n\mathcal{H}(\mathbf{vSUS}) + \mathcal{O}(\mathbf{nSUS} \lg n) + o(n)$*

bits. It can be computed in time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vSUS})))$, and supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1 + \lg \mathbf{nSUS} / \lg \lg n)$ for any value of $i \in [1..n]$. If i is chosen uniformly at random in $[1..n]$ the average query time is $\mathcal{O}(1 + \mathcal{H}(\mathbf{vSUS}) / \lg \lg n)$.

Proof. We first use Theorem 9 to find the SUS partition of optimal size \mathbf{nSUS} , and the corresponding vector \mathbf{vSUS} formed by the sizes of the subsequences of this partition. Then we apply Theorem 8: For the data structure S we use Theorem 5, whereas for P we use Theorem 4. Note $\mathcal{H}(\mathbf{vSUS})$ is both $\mathcal{H}_0(S)$ and $\mathcal{H}(\mathbf{vRuns})$ for permutation π' . The result follows immediately. \square

One would be tempted to consider the case of a permutation π covered by \mathbf{nSUS} upsequences which form strict runs, as a particular case. Yet, this is achieved by resorting directly to Theorem 4. The corollary extends verbatim to shuffled monotone sequences.

Corollary 11 *There is an encoding scheme using at most $n\mathcal{H}(\mathbf{vSUS}) + \mathcal{O}(\mathbf{nSUS} \lg n) + o(n)$ bits to encode a permutation π over $[1..n]$ optimally covered by \mathbf{nSUS} shuffled upsequences, of lengths forming the vector \mathbf{vSUS} , and made up of strict runs. It can be built within time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vSUS}) / \lg \lg n))$, and supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1 + \lg \mathbf{nSUS} / \lg \lg n)$ for any value of $i \in [1..n]$. If i is chosen uniformly at random in $[1..n]$ then the average query time is $\mathcal{O}(1 + \mathcal{H}(\mathbf{vSUS}) / \lg \lg n)$.*

Proof. It is sufficient to invert π and represent π^{-1} using Theorem 4, since in this case π^{-1} is covered by \mathbf{nSUS} ascending runs of lengths forming the vector \mathbf{vSUS} : If $i_0 < i_1 < \dots < i_m$ forms a strict upsequence, so that $\pi(i_t) = \pi(i_0) + t$, then calling $j_0 = \pi(i_0)$ we have the ascending run $\pi^{-1}(j_0 + t) = i_t$ for $0 \leq t \leq m$. \square

Once more, our construction translates into an improved sorting algorithm, improving on the complexity $\mathcal{O}(n(1 + \lg \mathbf{nSUS}))$ of the algorithm by Levopoulos and Petersson [LP94].

Corollary 12 *We can sort an array of length n , optimally covered by \mathbf{nSUS} shuffled upsequences, in time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vSUS})))$, where \mathbf{vSUS} are the lengths of the decomposition found by the algorithm of Theorem 9.*

Proof. Our construction in Theorem 10 finds and separates the subsequences of π , and sorts them, all within this time (we do not need to build the string S). \square

Open problem Note that the algorithm of Theorem 9 finds a partition of minimal size \mathbf{nSUS} (this is what we refer to with “optimally covered”), but that the entropy $\mathcal{H}(\mathbf{vSUS})$ of this partition is not necessarily minimal: There could be another partition, even of size larger than \mathbf{nSUS} , with lower entropy. Our results are only in function of the entropy of the partition of minimal size \mathbf{nSUS} found. This is unsatisfactory, as the ideal would be to speak in terms of the minimum possible $\mathcal{H}(\mathbf{vSUS})$, just as we could do for $\mathcal{H}(\mathbf{vRuns})$.

An example, consider the permutation $(1, 2, \dots, n/2-1, n, n/2, n/2+1, \dots, n-1)$, for some even integer n . The algorithm of Theorem 9 yields the partition $\{(1, 2, \dots, n/2-1, n), (n/2, n/2+1, \dots, n-1)\}$ of entropy $\mathcal{H}(\langle n/2, n/2 \rangle) = n \lg 2 = n$. This is suboptimal, as the partition $\{(1, 2, \dots, n/2-1, n/2, n/2+1, \dots, n-1), (n)\}$ is of much smaller entropy, $\mathcal{H}(\langle n-1, 1 \rangle) = (n-1) \lg \frac{n}{n-1} + \lg n = \mathcal{O}(\lg n)$.

On the other hand, a greedy online algorithm cannot minimize the entropy of a SUS partitioning. As an example consider the permutation $(2, 3, \dots, n/2, 1, n, n/2+1, \dots, n-1)$, for some even integer n . A greedy online algorithm that after processing a prefix of the sequence minimizes the entropy of such prefix, produces the partition $\{(1, n/2+1, \dots, n-1), (2, 3, \dots, n/2, n)\}$, of size 2 and entropy $\mathcal{H}(\langle n/2, n/2 \rangle) = n$. However, a much better partition is $\{(1, n), (2, 3, \dots, n-1)\}$, of size 2 and entropy $\mathcal{H}(\langle 2, n-2 \rangle) = \mathcal{O}(\lg n)$.

We doubt that the SUS partition minimizing $\mathcal{H}(\mathbf{vSUS})$ can be found within time $\mathcal{O}(n(1 + \mathcal{H}(\mathbf{vSUS})))$ or even $\mathcal{O}(n(1 + \lg \mathbf{nSUS}))$. Proving this right or wrong is an open challenge.

5.3 Shuffled Monotone Sequences

No efficient algorithm is known to compute the minimum number \mathbf{nSMS} of shuffled monotone sequences composing a permutation, let alone finding a partition minimizing the entropy $\mathcal{H}(\mathbf{vSMS})$ of the lengths of the subsequences. The problem is NP-hard, by reduction to the computation of the “cochromatic” number of the graph corresponding to the permutation [KSW96].

Yet, should such a partition into monotone subsequences be available, and be of smaller entropy than the partitions considered in the previous sections, this would yield an improved encoding by doing just as in Theorem 10 for SUS.

Note that it takes a difference by a superpolynomial margin between the values of \mathbf{nSUS} and \mathbf{nSMS} to yield a noticeable difference between $\lg \mathbf{nSUS}$ and $\lg \mathbf{nSMS}$, and hence between the values of $\mathcal{H}(\mathbf{vSUS})$ and $\mathcal{H}(\mathbf{vSMS})$. It seems unlikely that such a difference would justify the difference of computing time between the two types of partitions, also different by a superpolynomial margin to the best of current knowledge (i.e., if $P \neq NP$).

6 Conclusions

Relation between space and time Bentley and Yao [BY76] introduced a family of search algorithms adaptive to the position of the element sought (also known as the “unbounded search” problem) through the definition of a family of adaptive codes for unbounded integers, hence proving that the link between algorithms and encodings was not limited to the complexity lower bounds suggested by information theory. Such a relation between “time” and “space” can be found in other contexts: algorithms to merge two sets define an encoding for sets [AL09], and the binary results of the comparisons of any deterministic sorting algorithm in the comparison model yields an encoding of the permutation being sorted.

We have shown that some concepts originally defined for adaptive variants of the algorithm MergeSort, such as runs and shuffled sequences, are useful in terms of the compression of permutations, and conversely, that concepts originally defined for data compression, such as the entropy of the sets of run lengths, are a useful addition to the set of difficulty measures previously considered in the study of adaptive sorting algorithms.

Much more work is required to explore the application to the compression of permutations and strings of the many other measures of preorder introduced in the study of adaptive sorting algorithms. Figure 1 represents graphically some of those measures of presortedness (adding to those described by Moffat and Petersson [MP92], those described in this and other recent work [BFN11]) and a preorder on them based on optimality implication in terms of the number of comparison performed. This is relevant for the space of the corresponding permutation encodings, and for the

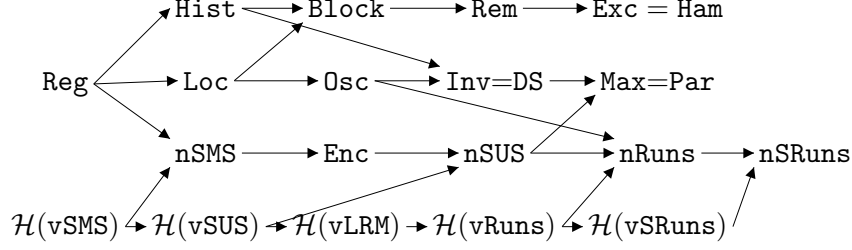


Figure 1: Partial order on some measures of disorder for adaptive sorting. New results are on the bottom line.

space used by the potential corresponding compressed data structures for permutations. Note that the reductions in this graph do not represent reductions in terms of optimality of the running time to find the partitions. For instance, we saw that $\mathcal{H}(\text{vSMS})$ -optimality implies $\mathcal{H}(\text{vSUS})$ -optimality in terms of the number of comparison performed, but not in terms of the running time. In terms of data structures, this relates to the construction time of the compressed data structure (as opposed to the space it takes).

Adaptive operators It is worth noticing that, in many cases, the time to support the operators on the compressed permutations is *smaller* as the permutation is more compressed, in opposition with the traditional setting where one needs to decompress part or all of the data in order to support the operators. This behavior, incidental in our study, is a very strong incentive to further develop the study of difficulty or compressibility measures: measures such that “easy” instances can both be compressed and manipulated in better time capture the essence of the data.

Compressed indices Interestingly enough, our encoding techniques for permutations compress both the permutation and its index (i.e., the extra data to speed up the operators). This is opposed to previous work [MRRR03] on the encoding of permutations, whose index size varied with the size of the cycles of the permutation, but whose data encoding was fixed; and to previous work [BHMR07] where the data itself can be compressed but not the index, to the point where the space used by the index dominates that used by the data itself. This direction of research is promising, as in practice it is more interesting to compress the whole succinct data structure or at least its index, rather than just the data.

Applications Permutations are everywhere, so that compressing their representation helps compress many other forms of data, and supporting in reasonable time the operators on permutations yield support for other operators.

As a first example, consider a natural language text tokenized into word identifiers. Its *word-based inverted index* stores for each distinct word the list of its occurrences in the tokenized text, in increasing order. This is a popular data structure for text indexing [BYRN11, WMB99]. By regarding the concatenation of the lists of occurrences of all the words, a permutation π is obtained that is formed by ν contiguous ascending runs, where ν is the vocabulary size of the text. The lengths of those runs corresponds to the frequencies of the words in the text. Therefore our representation achieves the zero-order word-based entropy of the text, which in practice compresses the text to about 25% of its original size [BCW90]. With $\pi(i)$ we can access any position of any inverted

list, and with $\pi^{-1}(j)$ we can find the word that is at any text position j . Thus the representation contains the text and its inverted index within the space of the compressed text.

A second example is given by compressed suffix arrays (CSAs), which are data structures for indexing general texts. A family of CSAs builds on a function called Ψ [GV06, Sad03, GGV03], which is actually a permutation. Much effort was spent in compressing Ψ to the zero- or higher-order entropy of the text while supporting direct access to it. It turns out that Ψ contains σ contiguous increasing runs, where σ is the alphabet size of the text, and that the run lengths correspond to the symbol frequencies. Thus our representation of Ψ would reach the zero-order entropy of the text. It supports not only access to Ψ but also to its inverse Ψ^{-1} , which enables so-called bidirectional indexes [RNOM09], which have several interesting properties. Furthermore, Ψ contains a number of strict ascending runs that depends on the high-order entropy of the text, and this allows compressing it further [NM07].

From a practical point of view, our encoding schemes are simple enough to be implemented. Some preliminary results on inverted indexes and compressed suffix arrays show good performances on practical data sets. As an external test, the techniques were successfully used to handle scalability problems in MPI applications [KMW10].

Followup Our preliminary results [BN09] have stimulated further research. This is just a glimpse of the work that lies ahead on this topic.

While developing, with J. Fischer, compressed indexes for Range Minimum Query indexes based on Left-to-Right Minima (LRM) trees [Fis10, SN10], we realized that LRM trees yield a technique to rearrange in linear time \mathbf{nRuns} contiguous ascending runs of lengths forming vector \mathbf{vRuns} , into a partition of $\mathbf{nLRM} = \mathbf{nRuns}$ ascending subsequences of lengths forming a new vector \mathbf{vLRM} , of smaller entropy $\mathcal{H}(\mathbf{vLRM}) \leq \mathcal{H}(\mathbf{vRuns})$ [BFN11]. Compared to a SUS partition, the LRM partition can have larger entropy, but it is much cheaper to compute and encode. We represent it on Figure 1 between $\mathcal{H}(\mathbf{vRuns})$ and $\mathcal{H}(\mathbf{vSUS})$.

While developing, with T. Gagie and Y. Nekrich, an elegant combination of previously known compressed string data structures to attain superior space/time trade-offs [BGNN10], we realized that this yields various compressed data structures for permutations π such that the times for $\pi()$ and $\pi^{-1}()$ are improved to log-logarithmic. While those results subsume our initial findings [BN09], the improved results now presented in Theorem 4 are incomparable, and in particular superior when the number of runs is polylogarithmic in n .

Acknowledgements

We thank Ian Munro, Ola Petersson and Alistair Moffat for interesting discussions.

References

- [AL09] Bruno T. Ávila and Eduardo S. Laber. Merge source coding. In *ISIT'09: Proceedings of the 2009 IEEE international conference on Symposium on Information Theory*, pages 214–218, Piscataway, NJ, USA, 2009. IEEE Press.

- [ANS06] D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 319–330, 2006.
- [Bae07] M. Baer. D-ary bounded-length Huffman coding. *CoRR*, abs/cs/0701012, 2007.
- [BCW90] T. Bell, J. Cleary, and I. Witten. *Text compression*. Prentice Hall, 1990.
- [BFN11] J  r  my Barbay, Johannes Fischer, and Gonzalo Navarro. LRM-Trees: Compressed indices, adaptive sorting, and compressed permutations. In *Proc. 22th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6661, pages 285–298, 2011.
- [BGNN10] J  r  my Barbay, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Alphabet partitioning for compressed rank/select and applications. In O. Cheong, K.-Y. Chwa, and K. Parks, editors, *Proceedings of ISAAC 2010*, LNCS, volume 6507, pages 315–326, 2010.
- [BHMR07] J  r  my Barbay, Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689. ACM, 2007.
- [BLNS09] N. Brisaboa, M. Luaces, G. Navarro, and D. Seco. A new point access method based on wavelet trees. In *Proc. 3rd International Workshop on Semantic and Conceptual Issues in GIS (SeCoGIS)*, LNCS 5833, pages 297–306, 2009.
- [BN09] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 111–122. Schloss Dagstuhl, Leibnitz Zentrum fuer Informatik, Germany, 2009.
- [BY76] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.
- [BYRN11] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011.
- [CHSV08] Y.-F. Chien, W.-K. Hon, R. Shah, and J. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proc. Data Compression Conference (DCC)*, pages 252–261, 2008.
- [CT91] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley, 1991.
- [Fis10] J. Fischer. Optimal succinctness for range minimum queries. In *Proc. 9th Symposium on Latin American Theoretical Informatics (LATIN)*, LNCS 6034, pages 158–169, 2010.
- [FMMN07] P. Ferragina, G. Manzini, V. M  kinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.
- [Fre75] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Math.*, 11:29–35, 1975.

- [FW94] M. Fredman and D. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and Systems Science*, 48(3):533–551, 1994.
- [GGG⁺07] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S.S. Rao. On the size of succinct indices. In *Proc. 15th Annual European Symposium on Algorithms (ESA)*, LNCS 4698, pages 371–382, 2007.
- [GGV03] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
- [GHSV06] A. Gupta, W.-K. Hon, R. Shah, and J.S. Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Proc. 16th Data Compression Conference (DCC)*, pages 213–222, 2006.
- [GMR06] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373. ACM, 2006.
- [Gol06] A. Golynski. Optimal lower bounds for rank and select indexes. In *Proc. 33th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4051, pages 370–381, 2006.
- [GRR08] A. Golynski, R. Raman, and S. Rao. On the redundancy of succinct data structures. In *Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, LNCS 5124, pages 148–159, 2008.
- [Gup07] A. Gupta. *Succinct Data Structures*. PhD thesis, Dept. of Computer Science, Duke University, 2007.
- [GV06] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.
- [Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1090–1101, 1952.
- [Kär99] J. Kärkkäinen. *Repetition-based text indexes*. PhD thesis, Dept. of Computer Science, University of Helsinki, Finland, 1999. Also available as Report A-1999-4, Series A.
- [KMW10] H. Kamal, S. Mirtaheeri, and A. Wagner. Scalability of communicators and groups in MPI. In *Proc. 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 264–275, 2010.
- [Knu98] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.
- [KSW96] André E. Kézdy, Hunter S. Snevily, and Chi Wang. Partitioning permutations into increasing and decreasing subsequences. *J. Comb. Theory Ser. A*, 73(2):353–359, 1996.

- [LP94] Christos Levcopoulos and Ola Petersson. Sorting shuffled monotone sequences. *Inf. Comput.*, 112(1):37–50, 1994.
- [Man85] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. In *IEEE Trans. Comput.*, volume 34, pages 318–325, 1985.
- [ML01] R. L. Milidiú and E. S. Laber. Bounding the inefficiency of length-restricted prefix codes. *Algorithmica*, 31(4):513–529, 2001.
- [MN07] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
- [MP92] Alistair Moffat and Ola Petersson. An overview of adaptive sorting. *Australian Computer Journal*, 24(2):70–77, 1992.
- [MR04] J. Ian Munro and S. Srinivasa Rao. Succinct representations of functions. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture Notes in Computer Science (LNCS)*, pages 1006–1015. Springer-Verlag, 2004.
- [MRRR03] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2719 of *Lecture Notes in Computer Science (LNCS)*, pages 345–356. Springer-Verlag, 2003.
- [MS76] J. Ian Munro and Philip M. Spira. Sorting and searching in multisets. *SIAM J. Comput.*, 5(1):1–8, 1976.
- [Mun96] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.
- [NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
- [Pǎ8] M. Pătraşcu. Succincter. In *Proc. 49th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.
- [RNOM09] L. Russo, G. Navarro, A. Oliveira, and P. Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.
- [RRR02] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k -ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.
- [Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
- [SN10] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.

- [ST85] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.
- [Wil83] D. Willard. Log-logarithmic worst case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17:81–84, 1983.
- [WMB99] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 2nd edition, 1999.